Parte CINCO
Exceções e Asserções

Capítulo DEZENOVE
Exceções

Objetivos do Exame

- Usar instruções try-catch e throw.
- Usar cláusulas catch, multi-catch e finally.
- Usar recursos Autoclose com uma instrução try-with-resources.
- Criar exceções personalizadas e recursos Autocloseable.

Exceção

Erros podem (e vão) acontecer em qualquer programa. Em Java, erros são representados por exceções.

Basicamente, em Java existem três tipos de exceção:

java.lang.Exception
Estende de java.lang.Throwable e representa erros que são esperados. Em alguns casos, o programa pode se recuperar deles. Alguns exemplos são: IOException, ParseException, SQLException.

java.lang.RuntimeException
Estende de java.lang.Exception e representa erros inesperados gerados em tempo de execução. Na maioria dos casos, o programa não consegue se recuperar deles. Alguns exemplos são: ArithmeticException, ClassCastException, NullPointerException.

java.lang.Error
Estende de java.lang.Throwable e representa problemas sérios ou condições anormais que um programa não deveria tratar. Alguns exemplos são: AssertionError, IOError, LinkageError, VirtualMachineError.

RuntimeException e suas subclasses não precisam ser capturadas, já que não são esperadas o tempo todo. Elas também são chamadas de *unchecked*.

Exception e suas subclasses (exceto RuntimeException) são conhecidas como *checked exceptions* porque o compilador precisa verificar se elas são capturadas em algum ponto por uma instrução try-catch.

Bloco Try-Catch

Existe apenas um bloco try:

```java
try {
   // Código que pode lançar uma exceção
} catch(Exception e) {
   // Fazer algo com a exceção usando
   // a referência e
}
```

Pode haver mais de um bloco catch (um para cada exceção a ser capturada).

Um bloco try é usado para englobar código que pode lançar uma exceção, não importa se é uma exceção verificada ou não.

Um bloco catch é usado para tratar uma exceção. Ele define o tipo da exceção e uma referência a ela.

Vejamos um exemplo:

```java
class Test {
    public static void main(String[] args) {
        int[] arr = new int[3];
        for(int i = 0; i <= arr.length; i++) {
            arr[i] = i * 2;
        }
        System.out.println("Done");
    }
}
```

Há um erro no programa acima, consegue ver?

Na última iteração do laço, i será 3, e como os arrays têm índices baseados em zero, uma exceção será lançada em tempo de execução:

```cpp
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3 at com.example.T
```

Se uma exceção não for tratada, a JVM fornece um tratador de exceções padrão que realiza as seguintes tarefas:

1. Imprime a descrição da exceção.
2. Imprime o rastreamento da pilha (hierarquia de métodos onde a exceção ocorreu).
3. Faz o programa terminar.

No entanto, se a exceção for tratada dentro de um bloco try-catch, o fluxo normal da aplicação é mantido e o restante do código é executado.

```java
class Test {
    public static void main(String[] args) {
        try {
            int[] arr = new int[3];
            for(int i = 0; i <= arr.length; i++) {
               arr[i] = i * 2;
            }
        } catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("Exception caught");
        }
        System.out.println("Done");
    }
}
```

A saída:

```php
Exception caught
Done
```

Este é um exemplo de exceção *unchecked*. Novamente, elas não precisam ser capturadas, mas capturá-las certamente é útil.

Por outro lado, temos exceções *checked*, que precisam ser envolvidas por um bloco try se você não quiser que o compilador reclame. Assim, este trecho de código:

```java
SimpleDateFormat sdf = new SimpleDateFormat("MM/dd");
Date date = sdf.parse("01-10"); // Erro de compilação
System.out.println(date);
```

Torna-se este:

```java
try {
    SimpleDateFormat sdf = new SimpleDateFormat("MM/dd");
    Date date = sdf.parse("01-10");
    System.out.println(date);
} catch (ParseException e) {
    System.out.println("ParseException caught");
}
```

Já que, de acordo com sua assinatura, o método parse lança uma java.text.ParseException (que estende diretamente de java.lang.Exception):

```java
public Date parse(String source) throws ParseException
```

A palavra-chave throws indica as exceções que um método pode lançar. Apenas as exceções *checked* precisam ser declaradas desta forma.

Agora, lembre-se de não confundir throws com throw. Este último de fato lançará uma exceção:

```java
public void myMethod() throws SQLException {
    throw new SQLException();
}
```

Também podemos capturar a superclasse diretamente:

```java
try {
    SimpleDateFormat sdf = new SimpleDateFormat("MM/dd");
    Date date = sdf.parse("01-10");
    System.out.println(date);
} catch (Exception e) {
    System.out.println("Exception caught");
}
```

Embora isso não seja recomendado, pois o bloco catch acima capturará toda exceção (verificada ou não) que possa ser lançada pelo código.

Então é melhor capturar ambas desta forma:

```java
try {
    SimpleDateFormat sdf = new SimpleDateFormat("MM/dd");
    Date date = sdf.parse("01-10");
    System.out.println(date);
} catch (ParseException e) {
    System.out.println("ParseException caught");
} catch (Exception e) {
    System.out.println("Exception caught");
}
```

Se uma exceção puder ser capturada em mais de um bloco, ela será capturada no primeiro bloco definido.

No entanto, devemos respeitar a hierarquia das classes: se uma superclasse for definida antes de uma subclasse, será gerado um erro de compilação:

```java
try {
    SimpleDateFormat sdf = new SimpleDateFormat("MM/dd");
    Date date = sdf.parse("01-10");
    System.out.println(date);
} catch (Exception e) {
    System.out.println("Exception caught");
} catch (ParseException e) {
    System.out.println("ParseException caught");
}
```

Um erro também é gerado se um bloco catch for definido para uma exceção que não poderia ser lançada pelo código dentro do bloco try:

```java
try {
    SimpleDateFormat sdf = new SimpleDateFormat("MM/dd");
    Date date = sdf.parse("01-10");
    System.out.println(date);
} catch (SQLException e) { // Erro de compilação
    System.out.println("ParseException caught");
}
```

O motivo desses dois erros é que o código de ambos os blocos catch nunca será executado (é inalcançável, como o compilador diz).

Em um caso, o bloco catch com a superclasse será executado para todas as exceções que pertencem àquele tipo, e no outro caso, a exceção nunca poderá ser lançada e o bloco catch nunca poderá ser executado.

Finalmente, se o código que lança uma exceção verificada não estiver dentro de um bloco try-catch, o método que contém esse código deve declarar a exceção na cláusula throws.

Nesse caso, o chamador do método deve capturar a exceção ou também declará-la na cláusula throws, e assim por diante, até que o método main do programa seja alcançado:

```java
public class Test {
    public static void main(String[] args)
            throws ParseException {
        m1();
    }
    private static void m1() throws ParseException {
        m2();
    }
    private static void m2() throws ParseException {
        m3();
    }
    private static void m3() throws ParseException {
        m4();
    }
    private static void m4() throws ParseException {
        SimpleDateFormat sdf =
             new SimpleDateFormat("MM/dd");
        Date date = sdf.parse("01-10");
        System.out.println(date);
    }
}
```

Multi-Catch

```java
try {
    // Código que pode lançar uma ou
    // duas exceções
} catch(Exception1 | Exception2 e) {
    // Fazer algo com a exceção capturada
    // usando a referência e
}
```

Captura Exception1 ou Exception2.

Finally

```java
try {
    // Código que pode lançar uma
    // exceção
} finally {
    // Bloco que é sempre executado
}
```

O bloco catch é opcional. Você pode ter ambos, ou apenas o bloco catch, ou apenas o bloco finally.

O bloco finally é sempre executado, não importa se uma exceção é lançada no bloco try, relançada dentro do bloco catch, ou não capturada de forma alguma.

Multi-Catch e Finally

Considere algo como o seguinte código:

```java
int res = 0;
try {
    int[] arr = new int[2];
    res = (arr[1] != 0) ? 10 / arr[1] : 10 * arr[2];
} catch (ArithmeticException e) {
    e.printStackTrace();
    return res;
} catch (IndexOutOfBoundsException e) {
    e.printStackTrace();
    return res;
}
return res;
```

Isso não é feio? Quero dizer, ter dois blocos catch com o mesmo código. Felizmente, o bloco *multi-catch* permite capturar duas ou mais exceções com um único bloco catch:

```java
try {
    ...
} catch (ArithmeticException | IndexOutOfBoundsException e) {
    e.printStackTrace();
    return res;
}
```

Pense no caractere pipe (|) como um operador OR. Além disso, note que há apenas uma variável ao final da cláusula catch para todas as exceções declaradas. Se você quiser diferenciar entre exceções, pode usar o operador instanceof:

```java
try {
   ...
} catch (ArithmeticException | IndexOutOfBoundsException e) {
    if(e instanceof ArithmeticException) {
        // Fazer algo diferente se o tipo da exceção
        // for ArithmeticException
    }
    e.printStackTrace();
    return res;
}
```

Além disso, a variável é tratada como final, o que significa que você não pode reatribuir (por que faria isso, afinal?):

```java
try {
    ...
} catch (ArithmeticException | IndexOutOfBoundsException e) {
    if(e instanceof ArithmeticException) {
        // Erro de compilação
        e = new ArithmeticException("Minha Exceção");
    }
} catch(Exception e) {
    e = new Exception("Minha Exceção"); // Compila!
    throw e;
}
```

Uma última regra. Você não pode combinar subclasses e suas superclasses no mesmo bloco *multi-catch*:

```java
try {
    ...
} catch (ArithmeticException | RuntimeException e) {
    // A linha acima gera erro de compilação
    // porque ArithmeticException é uma subclasse de
    // RuntimeException
    e.printStackTrace();
    return res;
}
```

Isso é semelhante ao caso em que uma superclasse é declarada em um bloco catch antes da subclasse. O código se torna redundante — a superclasse sempre capturará a exceção.

Voltando a este trecho de código:

```java
int res = 0;
try {
    int[] arr = new int[2];
    res = (arr[1] != 0) ? 10 / arr[1] : 10 * arr[2];
} catch (ArithmeticException | IndexOutOfBoundsException e) {
    e.printStackTrace();
    return res;
}
return res;
```

Já que o valor de res é sempre retornado, podemos usar um bloco finally:

```java
try {
    ...
} catch (ArithmeticException | IndexOutOfBoundsException e) {
    e.printStackTrace();
} finally {
    return res;
}
```

O bloco finally é SEMPRE executado, mesmo quando uma exceção é capturada ou quando o bloco try ou catch contém uma instrução return. Por esse motivo, ele é comumente usado para fechar recursos como conexões com banco de dados ou manipuladores de arquivos.

Há apenas uma exceção a essa regra. Se você chamar System.exit(), o programa terminará anormalmente sem executar o bloco finally. No entanto, como é considerado má prática chamar System.exit(), isso raramente acontece.

Try-With-Resources

Recurso que é fechado automaticamente

```java
try (AutoCloseableResource r = new AutoCloseableResource()) {
    // Código que pode lançar uma exceção
} catch(Exception e) {
    // Lidar com a exceção
} finally {
    // Sempre executa
}
```

O recurso é fechado após o bloco try terminar.

Os blocos catch e finally são ambos opcionais em um try-with-resources.

---

try-with-resources

Como dissemos antes, o bloco finally geralmente é usado para fechar recursos. Desde o Java 7, temos o bloco try-with-resources, no qual, dentro do bloco try, um ou mais recursos são declarados para que possam ser fechados sem fazê-lo explicitamente em um bloco finally:

```java
try (BufferedReader br =
       new BufferedReader(new FileReader("/file.txt"))) {
    int value = 0;
    while((value = br.read()) != -1) {
        System.out.println((char)value);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

No exemplo, o BufferedReader é fechado após o bloco try finalizar sua execução. Isso seria equivalente a:

```java
BufferedReader br = null;
try {
    int value = 0;
    br = new BufferedReader(new FileReader("/file.txt"));
    while((value = br.read()) != -1) {
        System.out.println((char)value);
    }
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        if (br != null) br.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Se você declarar mais de um recurso, eles devem ser separados por ponto e vírgula:

```java
try (FileReader fr = new FileReader("/file.txt");
     BufferedReader br = new BufferedReader(fr)) {
    ...
}
```

Além disso, recursos declarados dentro de um bloco try-with-resources não podem ser usados fora desse bloco (primeira razão: estão fora do escopo; segunda razão: foram fechados após o término do bloco try):

```java
try (BufferedReader br =
        new BufferedReader(new FileReader("/file.txt"))) {
    ...
}
String line = br.readLine(); // Erro de compilação
```

Agora, não pense que qualquer classe funcionará em um try-with-resources.

```java
class MyResource {
    void useResource() { }
}
...
try (MyResource r = new MyResource()) { // Erro de compilação
    r.useResource();
}
```

As classe(s) usadas em um bloco try-with-resources devem implementar uma das seguintes interfaces:

- java.lang.AutoCloseable
- java.io.Closeable

Ambas declaram um método close(), e a única diferença prática entre essas duas interfaces é que o método close da interface Closeable apenas lança exceções do tipo IOException:

```java
void close() throws IOException;
```

Enquanto o método close() da interface AutoCloseable lança exceções do tipo Exception (em outras palavras, pode lançar praticamente qualquer tipo de exceção):

```java
void close() throws Exception;
```

Assim, o método close() é chamado automaticamente e, se esse método realmente lançar uma exceção, podemos capturá-la no bloco catch.

```java
class MyResource implements AutoCloseable {
    public void close() throws Exception {
        int x = 0;
        //...
        if(x == 1) throw new Exception("Close Exception");
    }
    void useResource() {}
}
...
try (Resource r = new Resource()) { // Problema resolvido!
    r.useResource();
} catch (Exception e) {
    e.printStackTrace();
}
```

Mas o que acontece se o bloco try também lançar uma exceção?

Bem, o resultado é que a exceção do bloco try “vence” e as exceções do método close() são “suprimidas”.

Na verdade, você pode recuperar essas exceções suprimidas chamando o método
Throwable[] java.lang.Throwable.getSuppressed()
a partir da exceção lançada pelo bloco try.

```java
try (Resource r = new Resource()) {
    r.useResource();
    throw new Exception("Exception inside try");
} catch (Exception e) {
    System.out.println(e.getMessage());
    Stream.of(e.getSuppressed())
        .forEach(t -> System.out.println(t.getMessage()));
}
```

A saída (assumindo que o método close() lance uma exceção):

```php
Exception inside try
Close Exception
```

Exceções personalizadas

Como exceções são classes, podemos simplesmente estender qualquer exceção da linguagem para criar nossas próprias exceções.

Se você quiser forçar o tratamento da sua exceção, estenda Exception ou uma de suas subclasses. Se você não quiser forçar isso, estenda RuntimeException ou uma de suas subclasses.

```java
class TooHardException extends Exception {
    public TooHardException(Exception e) {
        super(e);
    }
}
class TooEasyException extends RuntimeException { }
```

Como você pode ver, é uma convenção adicionar Exception ao nome das suas classes. As classes Error e Throwable não são usadas para exceções personalizadas.

Os principais membros da classe Exception que você deve conhecer são:

Descrição	Construtor/Método
Construtor padrão	Exception()
Construtor que recebe uma mensagem	Exception(String)
Construtor que recebe outra exceção (que representa a causa)	Exception(Throwable)
Retorna a mensagem da exceção	String getMessage()
Retorna (se houver) a causa da exceção	Throwable getCause()
Retorna a lista de exceções suprimidas	Throwable[] getSuppressed()
Imprime o rastreamento da pilha (incluindo causa e exceções suprimidas)	void printStackTrace()

Pontos-chave

- Em Java, existem três tipos de exceção:
    - java.lang.Exception
    - java.lang.RuntimeException
    - java.lang.Error
- RuntimeException e suas subclasses não precisam ser capturadas, pois não são esperadas o tempo todo. Elas também são chamadas de *unchecked* (não verificadas).
- Exception e suas subclasses (exceto RuntimeException) são conhecidas como exceções verificadas (*checked exceptions*) porque o compilador precisa verificar se elas são capturadas em algum ponto por uma instrução try-catch.
- Se uma exceção puder ser capturada em mais de um bloco, a exceção será capturada no primeiro bloco definido.
- No entanto, devemos respeitar a hierarquia das classes. Se uma superclasse for definida antes de uma subclasse, será gerado um erro de compilação.
- Se o código que lança uma exceção verificada não estiver dentro de um bloco try-catch, o método que contém esse código deve declarar a exceção na cláusula throws.
- Nesse caso, o chamador do método deve capturar a exceção ou também declará-la na cláusula throws, e assim por diante, até o método main do programa ser alcançado.
- O bloco *multi-catch* permite capturar duas ou mais exceções não relacionadas com um único bloco catch:

- O bloco finally é SEMPRE executado, mesmo quando uma exceção é capturada ou quando o bloco try ou catch contém uma instrução return.
- Em um bloco try-with-resources, um ou mais recursos são declarados para que possam ser fechados automaticamente após o término do bloco try, bastando implementar:
    - java.lang.AutoCloseable ou
    - java.io.Closeable:
    -
```java
try (Resource r = new Resource()) {
    //...
} catch(Exception e) { }
```
- Ao usar um bloco try-with-resources, catch e finally são opcionais.
- Você pode criar suas próprias exceções apenas estendendo java.lang.Exception (para exceções verificadas) ou java.lang.RuntimeException (para não verificadas).

Autoavaliação

1. Dado:

```java
public class Question_19_1 {
    protected static int m1() {
        try {
            throw new RuntimeException();
        } catch(RuntimeException e) {
             return 1;
        } finally {
             return 2;
        }
    }
    public static void main(String[] args) {
        System.out.println(m1());
    }
}
```

Qual é o resultado?
A. 1
B. 2
C. A compilação falha
D. Uma exceção ocorre em tempo de execução

---

2. Dado:

```java
public class Question_19_2 {
    public static void main(String[] args) {
        try {
            // Não faz nada
        } finally {
            // Não faz nada
        }
    }
}
```

Qual das seguintes afirmações é verdadeira?
A. O código não compila corretamente
B. O código compilaria corretamente se adicionássemos um bloco catch
C. O código compilaria corretamente se removêssemos o bloco finally
D. O código compila corretamente como está

---

3. Quais das seguintes afirmações são verdadeiras?
A. Em um try-with-resources, o bloco catch é obrigatório.
B. A palavra-chave throws é usada para lançar uma exceção.
C. Em um bloco try-with-resources, se você declarar mais de um recurso, eles devem ser separados por ponto e vírgula.
D. Se um bloco catch for definido para uma exceção que não pode ser lançada pelo código no bloco try, será gerado um erro de compilação.

---

4. Dado:

java Copiar Editar

```java
class Connection implements java.io.Closeable {
    public void close() throws IOException {
        throw new IOException("Close Exception");
    }
}

public class Question_19_4 {
    public static void main(String[] args) {
        try (Connection c = new Connection()) {
            throw new RuntimeException("RuntimeException");
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Qual é o resultado?
A. Close Exception
B. RuntimeException
C. RuntimeException e então Close Exception
D. A compilação falha
E. O rastreamento da pilha de uma exceção não capturada é impresso

---

5. Quais das seguintes exceções são subclasses diretas de RuntimeException?
A. java.io.FileNotFoundException
B. java.lang.ArithmeticException
C. java.lang.ClassCastException
D. java.lang.InterruptedException